NUMA-Aware Graph Mining Techniques for Performance and Energy Efficiency

Michael Frasca, Kamesh Madduri, Padma Raghavan
Department of Computer Science and Engineering,
The Pennsylvania State University
University Park, Pennsylvania, USA
Email: {mrf218, madduri, raghavan}@cse.psu.edu

***Abstract*—We investigate dynamic methods to improve the power and performance profiles of large irregular applications on modern multi-core systems. In this context, we study a large sparse graph application, Betweenness Centrality, and focus on memory behavior as core count scales. We introduce new techniques to efficiently map the computational demands onto non-uniform memory architectures (NUMA). Our dynamic design adapts to hardware topology and dramatically improves both energy and performance. These gains are more significant at higher core counts.**

We implement a scheme for adaptive data layout, which reorganizes the graph after observing parallel access patterns, and a dynamic task scheduler that encourages shared data between neighboring cores. We measure performance and energy consumption on a modern multi-core machine and observe that mean execution time is reduced by 51.2% and energy is reduced by 52.4%.

I. Introduction

Large data applications are growing in size and scope [1], and system design innovations are trying to keep pace with application demands. In the context of unstructured algorithms, shared memory designs are a necessity as irregular domains induce complex dependencies that preclude effective domain decomposition [2]. Multi-core architectures and fine grain parallel paradigms will dictate future system and software designs [3], and we argue that aligning software with appropriate hardware features will enable high efficiencies.

Efficiency goals include both energy minimization and performance scalability under fixed problem size, and we analyze both of these aspects in the context of a heavily studied graph mining algorithm, Betweenness Centrality (BC) [4], [5], which serves to represent a broad class of sparse parallel computation. Power and energy concerns are a major driver of next generation designs [6], and scalability profiles must improve to justify many-core hardware investments. We know that irregular codes contain significant headroom for optimization, as they spend substantial amounts of time waiting for the memory hierarchy to produce data. Static power is wasted in both the CPU and memory domains to maintain state during idleness, and dynamic power is consumed to exercise auxiliary hardware structures. Removing this idleness will also reap performance benefits at high core counts.

Our specific concern is the growing disparity between computation and inter-core communication within multi-core architectures. Core count can grow exponentially while Moore's Law [7] holds true, and the memory hierarchy will similarly expand in size and complexity. Memory issues already dominate high-performance software design [8], but the many-core trend complicates this aspect. The distances between computational cores and memory units (on-chip caches and off-chip DRAM) will show great diversity within a system, otherwise known as Non-Uniform Memory Access (NUMA). NUMA topologies are unavoidable, and can drastically limit performance if exercised with abandon. We argue that designing parallel software with NUMA costs in mind is necessary, and should be a best-practice in high-performance software design. For irregular codes, e.g. graph mining, this challenge demands a dynamic approach, and we put forth a proposal for online optimizations that assert respect for NUMA costs. Tasks have affinity to threads that produced them, and load-balancing occurs in an efficient manner with respect to the current NUMA topology. As a result, neighboring threads work on tasks that are "neighbors" in terms of the current graph's connectivity.

II. Background

We first introduce background content to frame our study.

A. Non-Uniform Memory Access (NUMA)

Multi-core architectures require complex memory hierarchies to provide both low access latency, as well as a coherent view for all processing cores. On chip, there are significant distances between cores and each core has cache regions that are closer than others. These disparate distances introduce non-uniformity in cache latency, where local caches have the best performance [9]. Additionally, cores may be arranged in distinct CPU sockets, which creates even larger distances.

Figure 1 depicts the cache topology for a modern multi-core architecture, Intel's Westmere-EX [10]. This processor has 8 cores and supports 8 concurrent execution threads. Each core has private L1 and L2 caches, and the L3 cache is shared via a bi-directional interconnect. This L3 design provides necessary

This research was supported in part by the National Science Foundation through awards 0821527, 0963839, and 1017882.

SC12, November 10-16, 2012, Salt Lake
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

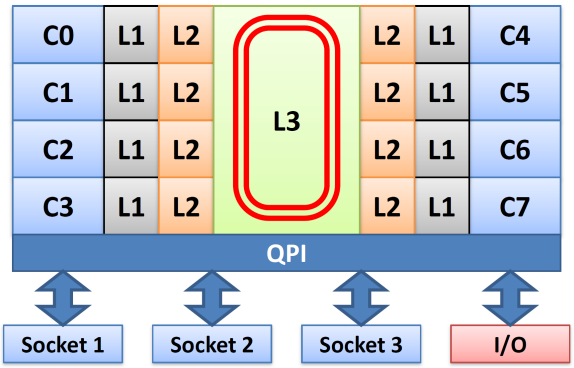

Fig. 1. NUMA Topology for Westmere-EX. Each processor has 8 cores with private L1 and L2 caches. The shared L3 cache is accessed via a bi-directional interconnect that exhibits NUMA. Each processor is directly connected to up to 3 other processor via QPI links, creating another NUMA layer.

scalability, but exhibits NUMA effects. L3 latency depends on the location of cache lines relative to each core. Additionally, multiple processors can form a shared multi-processing (SMP) system in which inter-processor communication occurs over direct Quick Path Interconnect (QPI) links. Data on remote memory banks adds another layer to the NUMA topology.

NUMA issues are generally the concern of the Operating System [9], [11], [12]. In this work, we acknowledge the complexity of NUMA issues within irregular algorithms and propose a more capable and dynamic runtime solution.

B. Betweenness Centrality

The Betweenness Centrality (BC) metric captures the importance of nodes in a graph, and concerns the ratio of all shortest paths to those that pass through a given node. We consider a state-of-the-art implementation [13] in our study, and aim to improve its performance by understanding and optimizing memory access behavior for NUMA architectures, e.g., multi- and many-cores.

The major computational component of BC is an all-pairs shortest path discovery. For each vertex, we must know all shortest paths to every other vertex in the graph. This is done by repeating an augmented breadth first search (BFS), shown in Algorithm 1, for each vertex in the graph. The BFS includes updates to vertex meta-data so that all shortest paths are collected. This depicts the code structure at a logical level, and we direct the reader to [13] for a complete technical description. For large graphs, one can select a sample of source vertices to estimate BC [14]–[16].

The BC algorithm is best suited for shared-memory architectures, as irregular graphs prohibit effective coarse-grain data partitioning. As a result, the BC implementation we consider uses OpenMP [17] to parallelize the loop found in line 7 of Algorithm 1. Because the cost of each loop iteration varies with respect to vertex edge counts, it is necessary to employ a dynamic schedule of tasks (vertices) to threads. Static task assignment leads to considerable workload imbalance. Efficient atomic intrinsics are used when updating vertex data, and locks on F are avoided with temporary thread-local arrays.

Algorithm 1 BFS procedure from source s

```
1:  procedure BC_BFS(G, s)
2:      d ← 0                                   ▷ Current BFS depth
3:      B ← {s}                                 ▷ Initial boundary set
4:      while |B| > 0 do
5:          F ← {}                              ▷ Frontier set
6:          d ← d + 1                           ▷ Increment depth
7:          for all v ∈ B in parallel do
8:              for all w ∈ G.adjacent(v) do
9:                  if w.depth = nil then       ▷ If unvisited
10:                     w.depth ← d             ▷ Mark depth
11:                     F.insert(w)             ▷ Add to frontier
12:                 end if
13:                 if w.depth = d then         ▷ If on the frontier
14:                     v.add_child(w)          ▷ Update paths from v
15:                     w.add_parent(v)         ▷ Update paths to w
16:                 end if
17:             end for
18:         end for
19:         B ← F                               ▷ Frontier becomes next boundary
20:     end while
21: end procedure
```

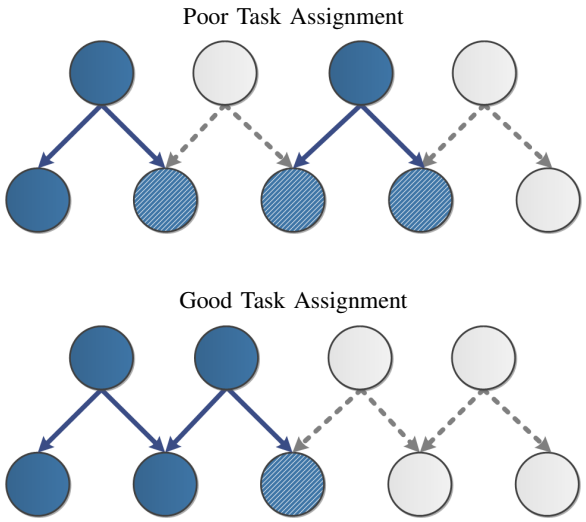

Fig. 2. Effects of task assignment with two threads (blue & white) and four boundary vertices. Unclustered boundary assignment (top) leads to increased per-thread working sets and inter-thread overlap on the frontier. An improved boundary assignment (bottom) creates more efficient frontier access behavior.

Each BFS is an iterative procedure, where the boundary B is processed and a new frontier F is discovered. As stated above, each iteration is parallelized by assigning boundary nodes to threads, and each thread processes their set to find new frontier nodes. A frontier node w may be "discovered" multiple times if multiple boundary nodes are adjacent. The check in line 9 (implemented with atomics) ensures that each node is added to the frontier only once, even though each path through w is recorded. Boundary nodes are dynamically assigned to threads to maintain workload balance, and our study considers the

effect that dynamic assignment has with respect to memory access behavior.

In Figure 2, we see an example graph section that is processed by two threads (blue & white). Each sub-figure shows a snapshot of a BFS search, where the top level depicts the boundary, and the neighboring frontier nodes are underneath. In the first example, the boundary vertices assigned to the blue thread are dispersed in terms of the graph topology, and this thread accesses 4 unique frontier vertices. Furthermore, the amount of overlap at the frontier with the white thread is quite high (blue and white striped vertices). The cost of shared vertices is more than just increased per-thread footprint. Frontier nodes are updated upon access (line 15 of Algorithm 1), which invalidates data in the cache hierarchy and causes cache thrashing between the threads. The second example's task assignment is much improved, where each thread accesses fewer frontier vertices and the inter-thread overlap is reduced.

We hypothesize that the effects just discussed will become increasingly significant as thread counts scale. On a many-core machine, working-set overlap could be large, inducing high cache coherence traffic. Some coherence is unavoidable, and we will aim to exploit a given NUMA topology to minimize the associated costs. A technique to cluster per-thread accesses will induce good locality within a thread, but also create significant relationships between the inter-thread working sets. Appropriate working-set affinity between neighboring threads will positively affect the distance non-local memory requests travel, and we expect meaningful optimizations to reap both energy and performance gains.

III. NUMA-Aware Runtime Optimizations

We design a novel workload scheduling approach that is general in its design, yet aims to mitigate memory issues as irregular applications scale. For BC, we expect irregular accesses to spread throughout the NUMA hierarchy, impeding performance and wasting energy. This behavior guides our approach to follow. We first discuss data layout before presenting our scheduling methodology.

A. Adaptive Data Layout (ADL)

Graph reordering is a common technique that increases both spatial and temporal locality of sparse data structures [18]. Heuristics, such as Reverse Cuthill-McKee (RCM) [19], search for useful graph permutations and are relatively cheap when amortized over the graph's lifetime. In the case of Betweenness Centrality, we can leverage the initial BFS to dynamically improve data locality. After completing one BFS, the graph is permuted based on the order in which graph nodes were discovered. This order is then reversed, for similar locality-based reasons that the Cuthill-McKee order is reversed. We term this strategy as *Adaptive Data Layout* (ADL). Logically, vertices are sorted by the tuple $\langle\ depth,\ time\ discovered\ \rangle$, where $depth$ is the first key to order by, and $time$ is secondary. After reordering the graph, spatial and temporal locality significantly increases, which yields significant cache advantages [20].

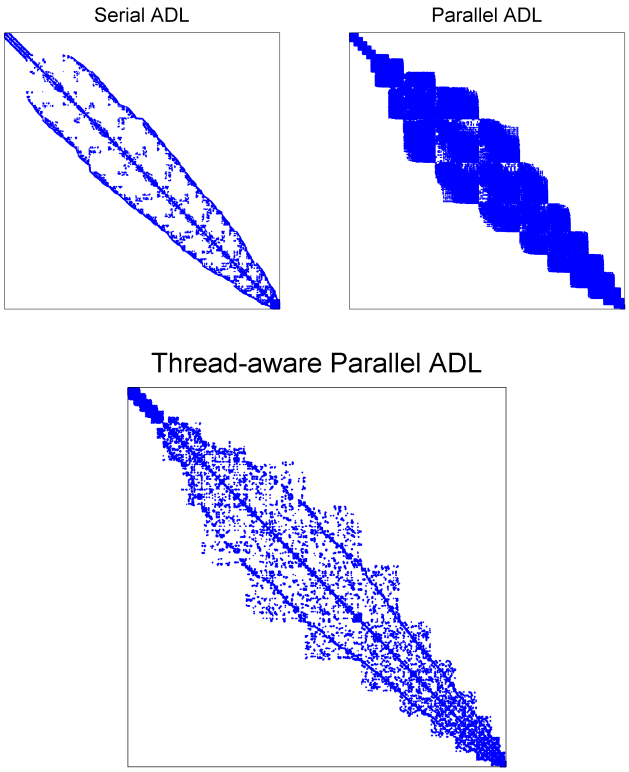

Fig. 3. Adjacency matrices for graph HB/bcsstk33 [21]. Graph reordering is an isomorphic transformation that modifies the sparse matrix representation while maintaining graph connectivity. ADL can create cache friendly data layouts when the BFS is run in serial. When parallel ADL order is applied, data layout degrades unless thread owner is incorporated (i.e. thread-aware).

When the BFS is parallelized, as is the case with our high-performance code, the precise order of discovery is suboptimal. Since node discovery is in parallel, the order of vertices by $time$ at a given depth is less structured and does not necessarily place neighboring vertices together. We realize that order is improved if we sort by the following tuple, $\langle\ depth,\ thread\ owner,\ time\ discovered\ \rangle$. This order produces better locality as the frontier sets associated with each vertex have a tighter layout.

Figure 3 presents adjacency matrices that depict the result of this Adaptive Data Layout (ADL). We run BC with a small graph from a standard benchmark collection [21] and observe the graph structure after applying a serial ADL order (top-left). To the right, we see that the graph layout is significantly worse in a naïve parallel BFS order. For most vertices, their respective edges are randomly distributed over large distances. When we include the thread owner in the permutation's sort, a significantly improved structure is observed (bottom). Connected vertices are more tightly clustered. This layout was inspired by BC's native access pattern, and we expect improved cache performance as a result. We examine the cost of ADL in Section V-C.

Algorithm 2 NUMA-Aware Work Queue: NUMA topology directs initial workload assignment and stealing pattern

```
1: procedure ProcessQueue(Q, Q_next)
2:     t_id ← GetThreadID()                          ▷ ID for the current thread
3:     for distance = 0 to Q.size()-1 do             ▷ For each work queue
4:         q_id ←NUMA_Neighbor(t_id, distance)       ▷ Get pointer to the next queue in topological order
5:         while task ← Q[q_id].next() do            ▷ For each remaining task
6:             tasks_new ← Process(task)             ▷ Process task and collect any new generated tasks
7:             Q_next[id].insert(tasks_new)          ▷ Add new tasks to thread-local queue
8:         end while
9:     end for
10:    return Q_next                                 ▷ Return generated task collection for the next iteration
11: end procedure
```

B. NUMA-Aware Workload Queues (NWQ)

Graph reordering greatly improves cache locality, but this may be lost as irregular codes scale. As discussed in Section II-B, growing parallelism will introduce significant overlap in per-thread working sets. When these data structures are updated, cache coherence traffic is a performance obstacle.

We propose a scheduler that maintains work queues per thread, and then uses these queues to distribute work in a NUMA-aware manner. In terms of BC, when a thread discovers a frontier node, the node is placed on its respective work queue. During the next BFS depth, tasks (i.e. vertices) are then scheduled dynamically to both create data locality and load balance. Each thread begins by processing its own queue, which maintains temporal locality across BFS phases. A completed thread then jumps to other queues and dynamically extracts work until all tasks are complete. To respect NUMA costs, we enforce that each thread walks through the remaining queues in order of increasing NUMA distance.

At a high level, our design mimics work stealing approaches in other domains. Work stealing traditionally allows idle threads to steal work at random [22], [23], or from whichever queue has the most remaining work [24]. We instead use NUMA topology to guide workload migration. Our technique (defined below) encourages overlapping working-sets among neighboring threads and reduces redundant data copies throughout disparate NUMA domains. It also reduces the average memory latency as cache misses due to invalidations or otherwise are likely satisfied by nearby NUMA regions.

A NUMA-respecting order is easily defined if we assume a topology that maps to a complete binary tree, e.g., at the lowest level threads {0,1},{2,3},{4,5},{6,7} are neighbors, and at the next level threads {0,1,2,3},{4,5,6,7} are topological neighborhoods. This example could represent a system of two 4-core processors. In such cases, the traversal for a thread $t_{id} \in (0, p-1)$ given p processors is computed via a binary exclusive-or operation below, where $distance \in (0, p-1)$.

$$\textbf{NUMA_Neighbor}(t_{id}, distance) = t_{id} \oplus distance$$

Algorithm 2 depicts the generic use of our NUMA-aware work queue. A queue is local to each thread, and each thread walks through the queues in topological order. As a result, each thread will work on their own local data first, and any thread that finishes early will process work from other threads in a cache-friendly manner. This procedure is implanted into the BC algorithm by representing the frontier/boundary as a

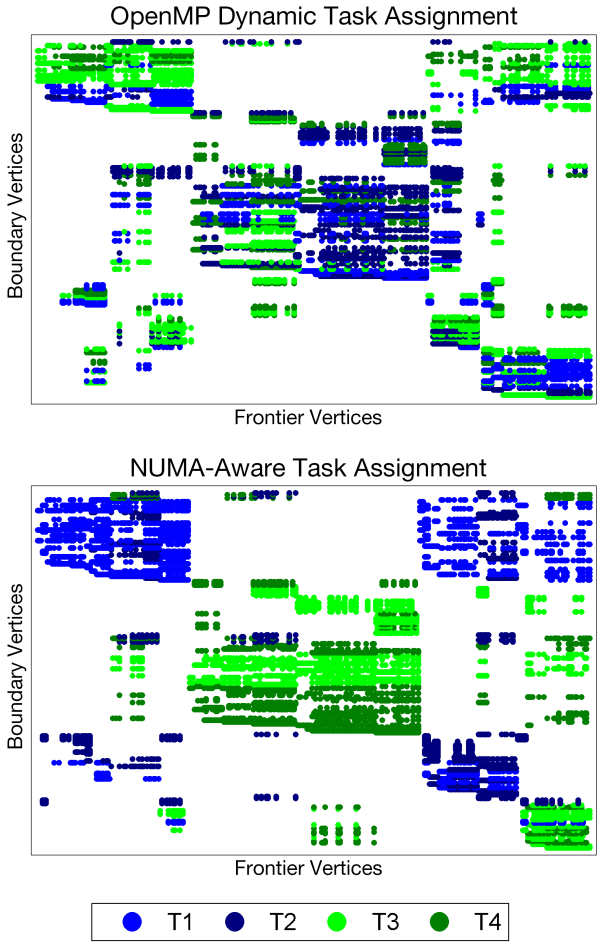

Fig. 4. Adjacency matrices for the fourth BFS front of HB/bcsstk33 [21] under OpenMP dynamic scheduling and NUMA-aware work queues. These represent a small portion of the graph seen in Figure 3. In both scenarios, boundary nodes are dynamically assigned to four threads (in chunks of 16), and the traversed frontier edges are identified by color. With the NUMA optimization, induced frontier node accesses have better clustering per thread and greater sharing of frontier nodes occurs between neighboring threads.

set of NUMA-aware work queues. Tasks have a one-to-one correspondence with boundary nodes, and processing a task (line 6) maps to visiting frontier nodes connected to a given boundary node (lines 8 through 17 of Algorithm 1). Threads extract vertices from the queues at a selected granularity to maintain performance, which is also done in the baseline code. In practice, we found that a task size of 16 vertices works well for our application and inputs.

Figure 4 depicts the effects of our scheduler on the same example matrix discussed before. We run the code with four threads and observe one BFS after applying the ADL layout. In the top example, we use the OpenMP dynamic scheduler (chunk size = 16), and on the bottom we use our NUMA-aware work queues. Threads 1 and 2 are neighbors, and threads 3 and 4 are neighbors with respect to NUMA distance. We view a portion of the graph (depth = 4), and the adjacency matrices show edges that connect the boundary/frontier at the time of discovery (which represent updates to the connected vertices). With the standard dynamic scheduler, there is no structure in the vertex to thread mapping, and we see that each thread updates a large number of frontier vertices. Additionally, most frontier nodes are updated by multiple threads. Our scheduler greatly improves this behavior, as the affinity encourages better clustering per thread. The scheduler influences how frontier nodes are accessed, and we see less sharing. Sharing that occurs is most often between NUMA neighbors.

In addition to cache benefits, this scheduling design has a better communication pattern than the centralized OpenMP dynamic scheduler. When parallelizing a for-loop, tasks are given to each thread at the granularity of a chunk size (e.g. 16 vertices), and when a thread completes its task, it returns to the scheduler for the next chunk. A critical bottleneck occurs as the act of obtaining the next chunk must be protected to avoid race conditions. If the number of threads is high, there is a good probability that threads wait for their turn at the scheduler. One can increase the chunk size, but the trade-off is a greater likelihood of workload imbalance. In our design, each work queue is independent and per queue locking is almost certain to generate fewer stalls.

IV. Experimental setup

We introduce the experimental setup employed to analyze our design above.

A. Graph Inputs

The input to BC is a large sparse graph that is either directed or undirected. We use two data sets maintained by the community that represent a diverse set of problem domains. From the data sets below, we use all graphs with more than 10 million vertices. All graphs are maintained by the University of Florida Spare Matrix collection [21], which we sourced from. Graph characteristics are listed in Table I.

DIMACS: The 10th DIMACS challenge was designed to further the development of large graph algorithms [25]. These graphs represent road networks, numerical simulation, and PDE domains, among others.

DIMACS			
Name	$\|V\|$	$\|E\|$	*Time / Source*
D1: germany_osm	11.5	12.4	5.7
D2: asia_osm	12.0	12.7	6.0
D3: hugetrace-00010	12.1	18.1	6.3
D4: road_central	14.1	16.9	6.7
D5: hugetrace-00020	16.0	24.0	8.5
D6: nlpkkt200	16.2	216.0	26.9
D7: rgg_n_2_24_s0	16.8	132.6	15.4
D8: delaunay_n24	16.8	50.3	10.7
D9: hugebubbles-00000	18.3	27.5	9.7
D10: hugebubbles-00010	19.5	29.2	10.4
D11: hugebubbles-00020	21.2	31.8	11.4
D12: road_usa	23.9	28.9	11.7
D13: nlpkkt240	28.0	373.2	49.4
D14: europe_osm	50.9	54.1	27.6

LAW			
Name	$\|V\|$	$\|E\|$	*Time / Source*
L1: uk-2002	18.5	292.2	7.9
L2: arabic-2005	22.7	631.2	13.6
L3: uk-2005	39.5	921.3	19.5
L4: it-2004	41.3	1135.7	29.8
L5: sk-2005	50.6	1930.3	43.6
L6: webbase-2001	118.1	992.8	36.7

TABLE I
Input Graphs. Vertex and Edge counts in millions. Baseline serial time per source vertex provided in seconds.

LAW: The Laboratory for Web Algorithmics (LAW) maintains a data set of large web crawls to research data compression techniques [26]. These graphs exhibit greater density and shorter diameters on average relative to the DIMACS graphs.

B. System

We investigate our design in the context of a modern multi-core system. This system is multi-socket and uses four 8-core Intel Xeon E7-8837 processors (Westmere-EX micro-architecture) and supports 32 concurrent threads. Each processor runs at 2.6 GHz, has a power envelope of 130 W, and is directly connected to the other three sockets via 3.2 GHz QPI links. Each core has 256 KB private L2 cache and shares a 24 MB NUMA L3 cache. The entire system has 512 GB of DRAM and aggregate memory bandwidth was observed to be 94.5 GB/s with the STREAM benchmark [27]. We use Intel's *icc* compiler, version 11.1, and apply *-fast* optimizations. Intel's OpenMP library 5.0 is used, thread affinity is set via *KMP_AFFINITY=compact*, and we set *KMP_LIBRARY=throughput*. In all cases, thread scheduling is set to *OMP_SCHEDULE=dynamic,16* which was experimentally found to be good on average across the input set.

C. Measurements

In our analysis, we measure execution time, energy consumption, and cache hit rates. Execution time is collected by instrumenting the main computational phase of BC (removing input load time). We measure total power draw between the wall and power supply, which is sampled once per second. We monitor cache events with PAPI [28] version 4.4.0. Time, energy, and cache events are measured on independent runs to minimize any instrumentation impact.

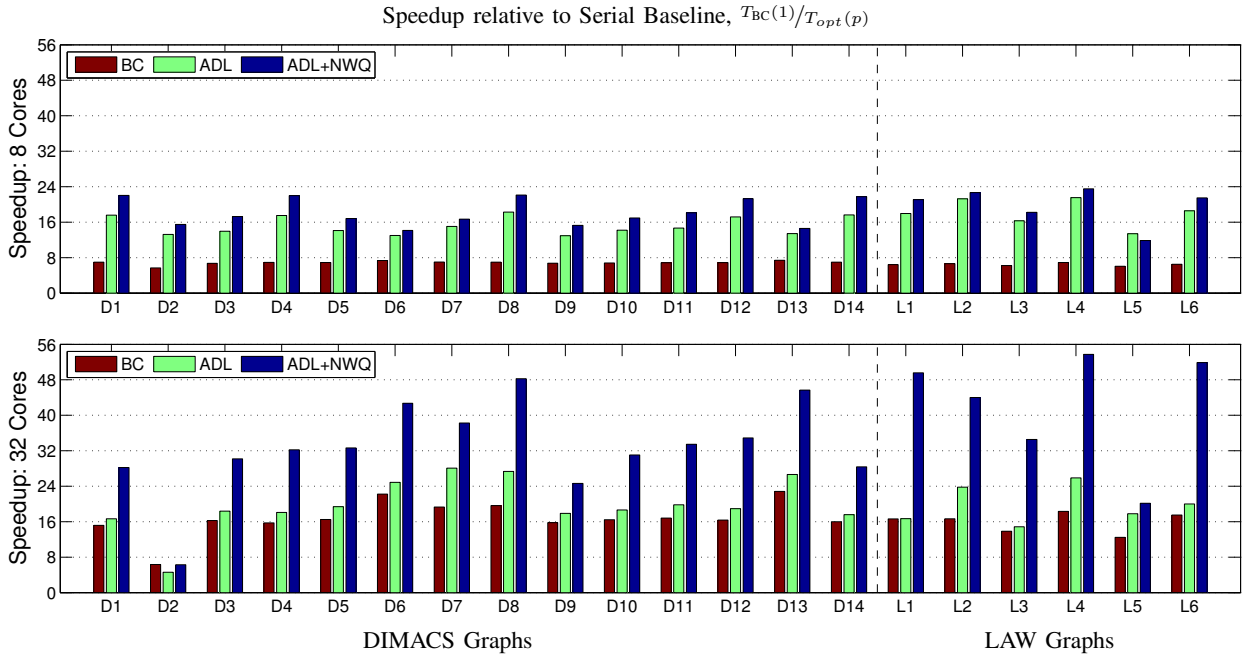

Fig. 5. Observed speedup relative to the serial BC baseline. Adaptive Data Layout (ADL) provides a majority of performance gains within socket (8 cores), but once multiple sockets are used, the NUMA effect limits performance. In this regime, the NUMA-aware work queues (NWQ) are responsible for most of the performance gains.

V. Results

We conduct our experiments on three BC variants and assume all inputs are provided with a random ordering. We observe the baseline version (BC), baseline with adaptive data layout (ADL), and NUMA-aware work queues in addition to data re-layout (ADL+NWQ). Using the affinity-queue implementation without adaptive data layout performed only 5% faster than the baseline on average, and we omit these results in interest of space.

Each configuration is run from serial to the maximum supported thread count, and all speedup measurements are relative to the serial baseline (BC at 1 thread). Aggregate results report the geometric mean. ADL reordering times are not included in the main results, as this cost would be amortized in practice. We examine reordering overhead separately in Section V-C.

Formally, we define speedup relative to the serial baseline. $T_{opt}(p)$ is the execution time with p cores, and opt is either BC, ADL, or ADL+NWQ.

$$\textbf{Speedup}(opt,\ p) = \frac{T_{\text{BC}}(1)}{T_{opt}(p)}$$

Power and energy consumption is measured, and we report the amount of energy saved relative to the baseline at the equivalent core count, $E_{\text{BC}}(p)$.

$$\textbf{Energy Reduction}(opt,\ p) = \frac{E_{\text{BC}}(p) - E_{opt}(p)}{E_{\text{BC}}(p)}$$

All runs select 50 random sources per graph to sample BC measurements, and the same 50 sources are used per graph for reproducibility across the experiments.

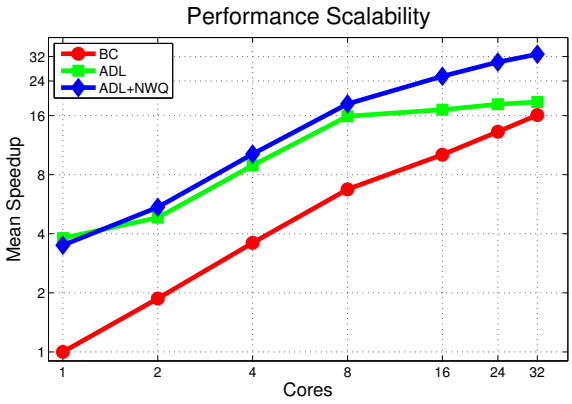

Fig. 6. Mean speedup versus $T_{\text{BC}}(1)$. Near linear scaling is observed for the baseline up to 8 cores, but speedup slows in the multi-socket regime. ADL yields large gains up to 8 cores, and NWQ adds only marginal benefit. However, NUMA effects limit ADL scaling and NWQ proves highly valuable as the core count increases.

A. Performance: Speedup and Scalability

We designed the NUMA-aware thread scheduler under the expectation that memory effects will only become more crucial as concurrency increases. As such, we first study the experimental data looking at the 8 and 32 thread runs. 8 threads saturate the cores on a single socket, and all four sockets are utilized at 32 threads. During multi-socket runs, caches and DRAM may be separated by QPI links, which dwarfs the distance scales within each chip. We expect the significance of NUMA issues to increase under these conditions.

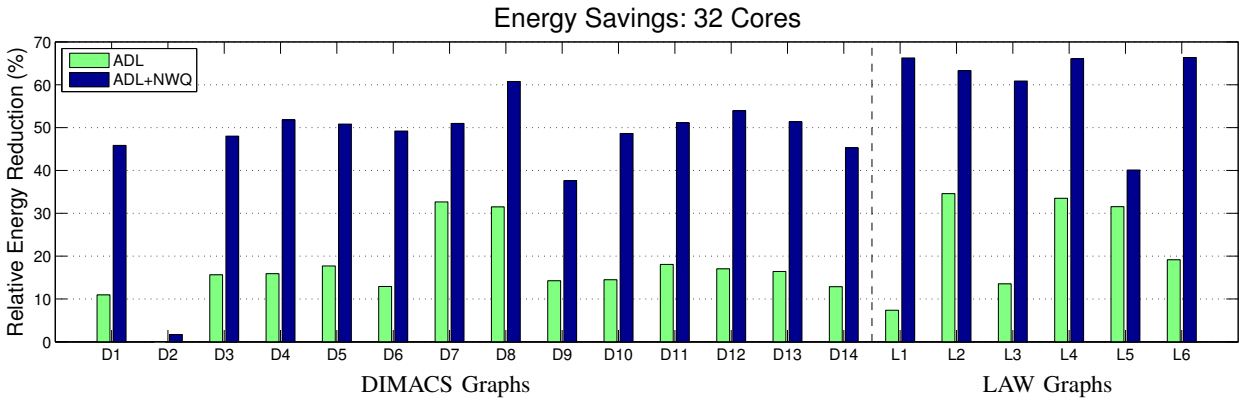

Fig. 7. Energy reduction observed relative to $E_{\text{BC}}(32)$. Many graphs exhibit a 50% or more reduction in energy consumed with the NUMA-aware work queues. A large portion of these gains comes from static energy saved by decreased execution times.

Figure 5 presents these speedup for each input at 8 and 32 threads. Graph inputs are sorted by increasing $|V|$, with the DIMACS graphs grouped together followed by the LAW graphs. At 8 threads, the data confirms that performance is affected by data layout, as our ADL reordering produces significant benefit. Our largest graph, webbase-2001 (L6), observes a speedup of 18.6x with ADL over the baseline serial code, while the baseline implementation achieves only 6.5x speedup. The NUMA-aware optimization (ADL+NWQ) provides only a modest improvement over ADL at 21.5x.

The performance behavior changes when all sockets and cores are enabled at 32 threads. For webbase-2001, baseline speedup is only 17.5x, while ADL manages to push speedup up to 20.0x. Here, we find that the NUMA-aware workload optimization is more valuable and speedup is 51.9x. In fact, the NUMA-aware optimization significantly enhances performance for most inputs under these conditions. *asia_osm* (D2) is the only one to show a performance loss. This graph has relatively small fronts and therefore low available parallelism. Excluding *asia_osm*, ADL+NWQ execution time is 51.2% less than BC on average.

Figure 6 presents aggregate results across all thread counts, and the trends become clear and convincing. At small core counts, ADL reorganizes the graph in a cache friendly manner, and large speedups are observed. Multiple sockets are used above 8 threads, and NUMA latency appears to limit performance for the baseline. The ADL variant suffers as well, and any gains are nearly surrendered at 32 threads. A driving motivation behind this work is the increased memory contention observed under high parallelism, and we suspect this to be the root cause poor scalability in spite of data layout optimization. Employing our NUMA-aware scheduler maintains the locality created by ADL, and we see the effects on performance as NUMA distances increase. At 32 threads, ADL+NWQ is 2.04x faster than BC on average.

These trends predict continued scalability challenges for irregular applications. Dynamic and locality preserving scheduling is shown to be an effective technique to mitigate these challenges and will be highly relevant in the many-core era.

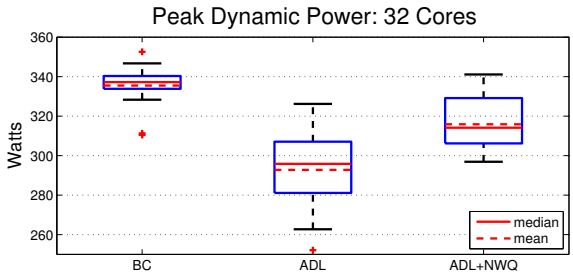

Fig. 8. Peak power observed across all graph inputs at 32 threads. Baseline BC code draws the most power, which is evidence that the memory hierarchy is doing more work for the same computation.

B. Power and Energy Consumption

We now consider the potential energy savings of our dynamic optimizations. There are two possible sources of energy reduction, i.e., static power and dynamic power. Static power is the baseline energy the system consumes when idle. Reducing application run-time directly reduces static power consumed. Dynamic power is the increase in power consumed above the static baseline and results from all actions that occur towards fulfilling application instructions. Among many others, arithmetic and logic units, super-scalar structures that re-order instructions, and cache accesses contribute to dynamic power. This may reduce if the system runs more efficiently, e.g., fewer caches levels traversed per memory operation, less instruction reordering if throughput remains high, or fewer pipeline flushes if branch instructions are more predictable. In our system, 662 Watts of static power is consumed, which is roughly ⅔ of total power when the system is at full load.

First, we examine total energy consumed for each input (the integral of static and dynamic power over time). Figure 7 presents the energy savings of ADL and ADL+NWQ relative to BC at 32 threads. These gains are generally correlated to observed speedup measurements, as static energy saved is a large fraction of total energy. Savings are large, except for *asia_osm* (ADL is -34.8%, not shown), which exhibited decreasing performance above 8 threads. Our largest graph,

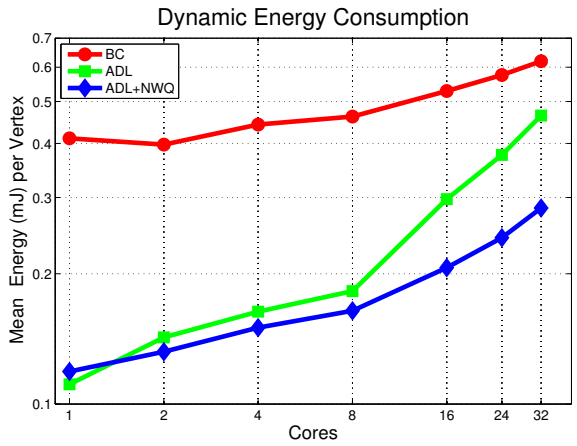

Fig. 9. Mean dynamic energy consumed per vertex across all 20 graphs. Energy naturally increases as more cores are employed, but the NUMA-aware work queues require significantly less energy at all concurrency levels.

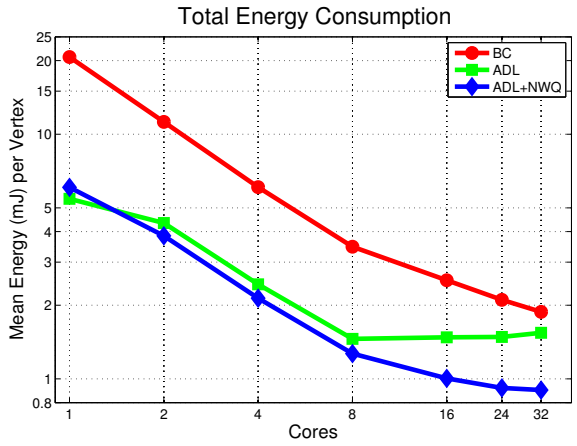

Fig. 10. Mean total energy consumed (dynamic + static) per vertex across all 20 graphs. Total energy decreases as a result of static energy saved. Without the NUMA-aware work queue, the ADL variant fails to achieve any gains above 8 threads. NWQ maintains energy reductions for the multi-socket configurations.

webbase-2001 (L6), shows energy reduction of 66.4% with ADL+NWQ over BC at 32 threads. This is largely a result of execution time being reduced from 105 seconds to 35.4 seconds. The mean savings (excluding *asia_osm*) is 17.9% for ADL and 52.4% for ADL+NWQ. These data show how important NUMA-aware optimizations are for large multi-core machines. It is possible to cut the energy budget in half through topology-aware task management.

We independently investigate dynamic power to measure the relative amount of resources each code requires. Figure 8 presents the peak power observed for each configuration at 32 threads. These box-plots show the four quartiles and any outliers across all 20 inputs. The baseline code draws the most power, which we suspect is due to the energy costs associated with frequent cache misses and coherence traffic. ADL+NWQ draws more power than ADL, and we do not have a straightforward explanation of this phenomenon. It may

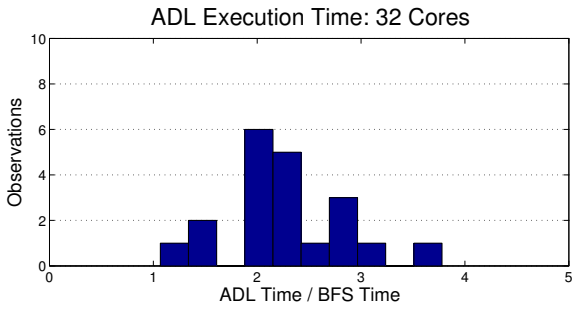

Fig. 11. Observed time to enable ADL relative to the cost of one BFS at 32 cores. For all graphs, the cost of ADL is only a few BFS traversals. This cost is easily amortized over the many sources analyzed in practice.

be that greater instruction level parallelism (ILP) is extracted due to fewer memory stalls, or less idleness occurs as threads acquire tasks from local NUMA work queues rather than a single global queue.

For the moment, we disregard static power and consider dynamic energy (the integral of dynamic power over time) as the measure of energy required to run the application. Figure 9 presents the mean energy consumed per graph when we normalize each observation by $|V|$. We consider this as a measure of energy per unit work. $|V|$ was chosen (opposed to $|E|$ or $|V| + |E|$) as a normalization factor since it is most highly correlated with the baseline execution time. Energy per unit work increases as more cores are added. This is expected, as parallel overheads consume energy. The baseline code is the biggest energy consumer, and the ADL variant quickly rises in energy above 8 threads. We interpret this as an indication of costly inter-processor communication and cache coherence. By adding the NUMA-aware work queues, we observe a much better trend line.

We again consider total energy consumed and view the scalability trends in Figure 10. The general trend is decreased energy per unit work as core count increases, which reflects static power savings as run time decreases. ADL deviates from the trend above 8 cores, and we see mean energy rise. Execution time does not decrease fast enough to offset rises in dynamic energy consumption. The energy performance is best under ADL+NWQ, which agrees with all of our previous results. Improved performance scalability and slow growth in dynamic energy translates into net energy savings. We have shown that these optimizations enable greater levels of power and performance efficiency, and can effectively decrease the workload costs.

C. Amortizing ADL Cost

Before examining detailed memory behavior, we verify that the cost of reorganizing data layout is justified by the observed performance gains. Betweenness Centrality requires a BFS search from every vertex to achieve the greatest precision. To avoid this cost, a subset of vertices can be used to obtain estimates. The graphs we consider have well over 10 million

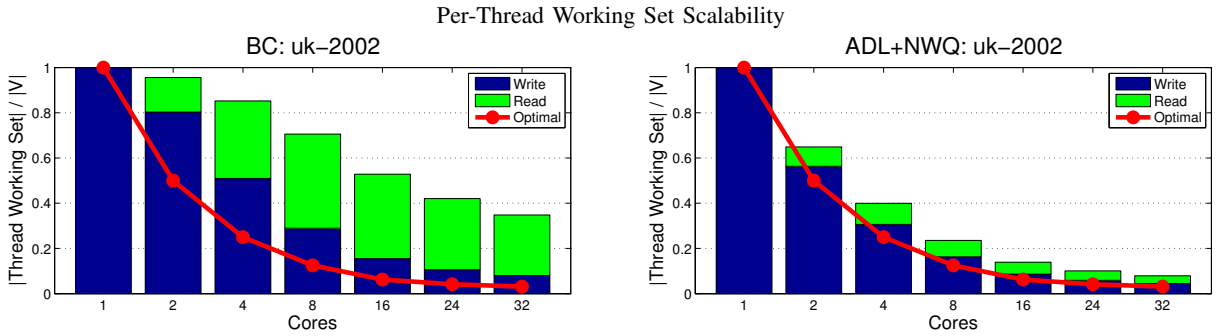

Fig. 12. Average per-thread working set size using graph uk-2002. In the baseline code, per-thread working sets remain high despite greater concurrency, while the ideal working set size is $|V|/p$. After NUMA optimizations, the average working set (i.e. vertices touched per thread) decreases dramatically.

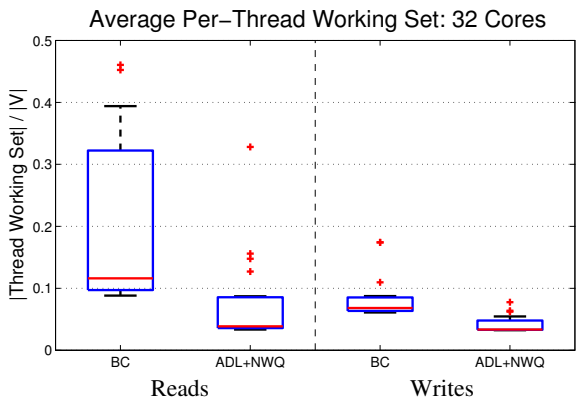

Fig. 13. Observed per-thread working sets for all inputs. Significantly fewer vertices are accessed per-thread after NUMA optimizations.

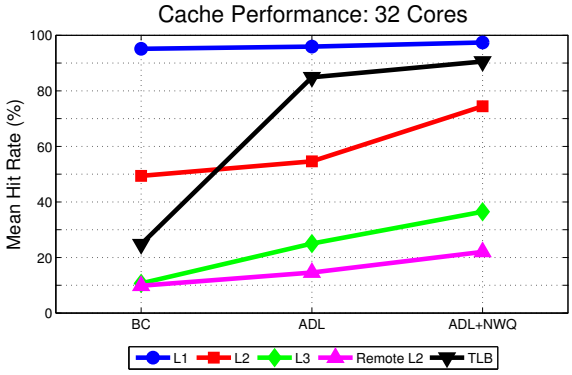

Fig. 14. Mean cache hit rates at 32 cores across all inputs. For each cache level, hit rates improve by successively adding ADL and NWQ optimizations. TLB and L3 hits rise considerably, which have significant performance impact.

vertices, and the above results considered executions that sample 50 random vertices. In practice, a much higher portion of vertices should be used [14]–[16]. Figure 11 shows the cost of running ADL versus the time of a single BFS search at 32 threads. The mean cost is 2.2x greater than a single BFS, which is insignificant in terms of the number of BFS searches required for high accuracy. Consequently, the cost of ADL is easily amortized over the runtime of BC.

NWQ overheads are contained within the dynamic task scheduling and included in the previous performance results. There are no other fixed costs associated with this method.

D. Relating Power and Performance to NUMA Behavior

We now analyze the results at a detailed algorithmic and architectural level to help explain the observed gains seen above. We previously argued that per-thread working set growth can impede performance at scale and that our NUMA-aware strategies can mitigate this. We expect this to manifest itself as improved cache performance and analyze both algorithmic and hardware features below.

Over the course of execution, each thread touches a portion of the graph. Edges found that connect the frontier induce vertex writes, while back edges only incur reads. The working set of each thread is the average number of vertices touched per BFS, and under ideal circumstances, the per-thread working set would be $|V|/p$, where p is the number of threads. Growth above this level results from working set overlap induced by the graph's connectivity but is exacerbated by poor thread assignment. We instrument our code to log the vertices that each thread processes while analyzing the web-crawl *uk-2002*. Figure 12 presents this data for the baseline code and ADL+NWQ as the thread count varies. As expected, per-thread working sets decrease, but the slow rate implies high working set overlap. After NUMA optimizations, each thread accesses significantly fewer unique vertices. This raises expectations of greater cache performance. In Figure 13, we show per-thread working set size for all inputs running 32 threads. We see that the same trend holds.

Cache performance is the physical manifestation of the memory behavior just discussed. On our processors, we collect L1, L2, L3, and TLB data via PAPI counters [28]. Additionally, cache hits found in remote L2 caches (on the same socket, but owned by another core) are also recorded. Figure 14 presents average cache performance when running 32 cores. Average performance improves across all cache levels as we employ our optimizations. Most notably, L3 and TLB hit rates significantly increase. Misses in either of these components create large memory latencies, as DRAM accesses are incurred. The remote L2 behavior is quite interesting, as it responds significantly to our NUMA optimization. When

the local L1, L2 or shared L3 cache fail to satisfy a memory request, there is increased probability that the data is in another L2 cache on the same socket. This is clear evidence that the NUMA-aware task affinity increases the sharing between neighboring threads. Higher L3 hit rates also point to the same conclusion. Neighboring threads share more data, which reduces the per socket footprint. Accordingly, shared L3 cache space is used more efficiently and performance improves.

VI. Related work

This work was made possible by the research of Madduri et al. [13], and their development of a lock-free Betweenness Centrality algorithm. Their design removed many communication bottlenecks, without which NUMA effects would not be of significance. Numerous other works have led to the current state-of-the-art in BC calculations [4], [5], [14]–[16].

NUMA issues have been studied in depth [9] and related optimizations are effective in many application domains such as dense linear algebra [29]. Despite this, NUMA is primarily the concern of operating systems (OS) through virtual memory techniques [11], [12], [30], [31]. Our level of integration between NUMA-aware features and the application goes further than an OS technique can provide.

Work stealing approaches are summarized well by Neill and Weirman [24], where they consider multiple polices for idle threads. In addition to stealing work at random or from threads with highest load, they discuss affinity-based approaches. Their approach requires models of cache affinity per task and is appropriate for coarse-grain task-based algorithm. Acar et al. also present locality aware techniques for workload stealing, but their implementation is also designed for task-based parallelism [32]. Betweenness Centrality exploits much finer grain parallelism, and to our knowledge, we are the first to incorporate affinity-aware scheduling in this domain.

Additionally, Min et al. study work affinity-aware work stealing approaches for distributed and shared memory systems [33]. Their shared memory design is most similar to ours, as threads traverse locality domains. Their evaluation only uses 8 threads and compares against heavier weight OpenMP tasking features.

VII. Conclusions and future work

In this work, we have developed and evaluated novel NUMA-aware optimizations for a large irregular application, Betweenness Centrality. First, we defined an Adaptive Data Layout (ADL) that observes parallel memory access behavior and dynamically reorders the graph to improve data locality. Independently, we introduced a NUMA-aware task scheduler that encourages better task to thread assignment, while considering the NUMA distances between threads.

In summary, we have shown though extensive evaluation on a modern multi-core system that power and performance scales well in the optimized code. We observe mean speedup of $2.04\times$ versus the baseline code at 32 threads, or a mean reduction in run-time by 51.2%. Mean energy was reduced by 52.4% at 32 threads.

References

[1] A. Hey, S. Tansley, and K. Tolle, *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA, 2009.

[2] J. Aspnes, J. Kirsch, and A. Krishnamurthy, "Load balancing and locality in range-queriable data structures," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM, 2004, pp. 115–124.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*. New York, NY, USA: ACM, 2008, pp. 1–15.

[4] U. Brandes, "A faster algorithm for betweenness centrality," *J. Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[5] ——, "On variants of shortest-path betweenness centrality and their generic computation," *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008.

[6] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale Computing Technology Challenges," *High Performance Computing for Computational Science–VECPAR 2010*, pp. 1–25, 2011.

[7] G. Moore *et al.*, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[8] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, March 1995.

[9] P. McCormick, R. Braithwaite, and W. Feng, "Empirical memory-access cost models in multicore numa architectures," Los Alamos National Laboratory (LANL), Tech. Rep., 2011.

[10] S. Sawant, U. Desai, G. Shamanna, L. Sharma, M. Ranade, A. Agarwal, S. Dakshinamurthy, and R. Narayanan, "A 32nm Westmere-EX Xeon® enterprise processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 2011, pp. 74–75.

[11] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for numa memory management," in *ACM SIGOPS Operating Systems Review*, vol. 23, no. 5. ACM, 1989, pp. 19–31.

[12] D. Tam, "Operating system management of shared caches on multicore processors," Ph.D. dissertation, University of Toronto, 2010.

[13] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[14] D. Eppstein and J. Wang, "Fast approximation of centrality," *Journal of Graph Algorithms and Applications*, vol. 8, no. 1, pp. 39–45, 2004.

[15] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proc. 5th Int'l. Workshop on Algorithms and Models for the Web-Graph (WAW 2007)*, ser. LNCS, A. Bonato and F. Chung, Eds., vol. 4863. Springer, Dec. 2007, pp. 124–137.

[16] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality," in *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX 2008)*. SIAM, Jan. 2008, pp. 90–100.

[17] L. Dagum and R. Menon, "Open MP: An Industry-Standard API for Shared-Memory Programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[18] A. Pinar and M. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. ACM, 1999, p. 30.

[19] W. Liu and A. Sherman, "Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices," *SIAM Journal on Numerical Analysis*, pp. 198–213, 1976.

[20] M. Frasca, A. Chatterjee, and P. Raghavan, "Can models of scientific software-hardware interactions be predictive?" *Procedia CS*, vol. 4, pp. 322–331, 2011.

[21] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[22] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. IEEE, 1994, pp. 356–368.

[23] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[24] D. Neill and A. Wierman, "On the benefits of work stealing in shared-memory multiprocessors," *Tech Report, Department of Computer Science, Carnegie Mellon University*, 2006.
[25] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "10th DIMACS implementation challenge-graph partitioning and graph clustering, 2011."
[26] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.
[27] J. McCalpin, "A survey of memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, pp. 19–25, 1995.
[28] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.
[29] W. Li and K. Pingali, "Access normalization: loop restructuring for numa computers," *ACM Transactions on Computer Systems (TOCS)*, vol. 11, no. 4, pp. 353–375, 1993.
[30] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 455–468.
[31] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai *et al.*, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 43–57.
[32] U. Acar, G. Blelloch, and R. Blumofe, "The data locality of work stealing," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2000, pp. 1–12.
[33] S. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.